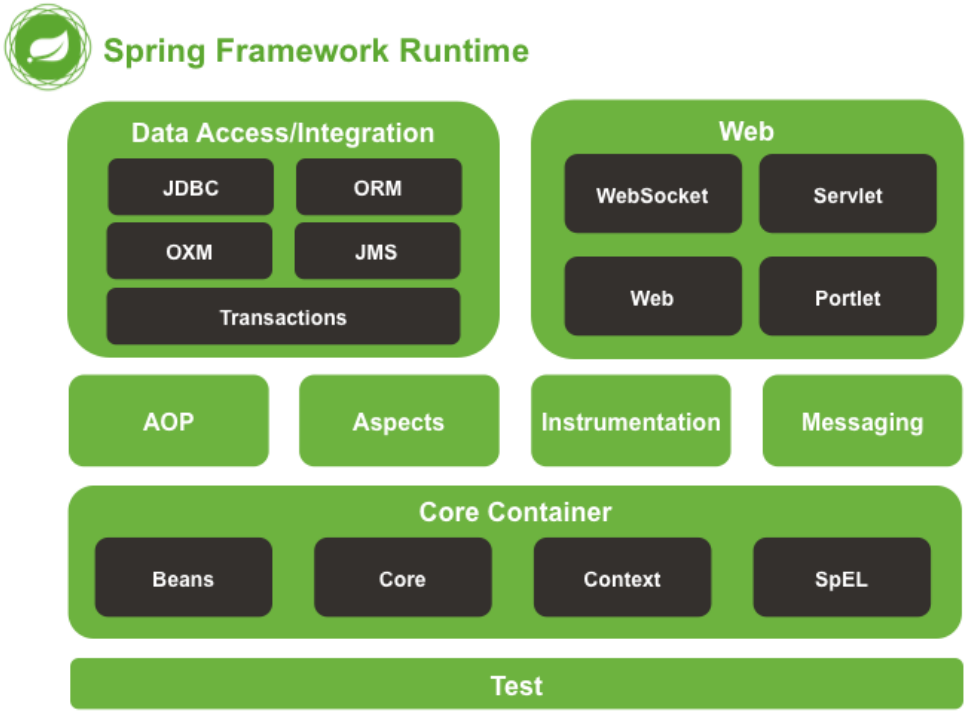
SPRING FRAMEWORKS



Capitulo 5

JDBC – PARTE I

Contenido

1 INTRODUCCIÓN

Los ejemplos de este capítulo están desarrollados con la base de datos EUREKABANK, el script para su creación lo puede obtener de la siguiente dirección web: https://github.com/gcoronelc/databases

2 DEFINICIÓN

El trabajo con el API JDBC es un poco engorroso, la Figura 1 muestra los pasos que se deben ejecutar para procesar una sentencia SQL con el API JDBC.

Proceso de Sentencias
1. Definir los parámetros de conexión
2. Abrir la conexión
3. Especificar la sentencia (consulta)
4. Preparar y ejecutar la sentencia
5. Preparar el ciclo e iterar los resultados (si los hay)
6. Realizar el trabajo para cada iteración
7. Procesar excepciones
8. Manejar las transacciones
9. Cerrar la conexión

Figura 1 Pasos para procesar una sentencia con el API JDBC.

Spring ofrece un framework para JDBC que abstrae su complejidad y hace más simple el procesamiento de una sentencia.

De los nueve pasos que se muestran en la Figura 1 cuando utilizamos el API JDBC, utilizando el framework Spring JDBC sólo los pasos 3 y 6 deben ser codificados por el programador, del resto de pasos se encarga el framework.

3 JERARQUÍA DE PAQUETES

Spring JDBC está formado por cuatro paquetes: core, dataSource, object y support.

- **org.springframework.jdbc.core**

  Contiene la clase JdbcTemplate.

- **org.springframework.datasource**

  Este es el principal acceso a las fuentes de datos de Spring JDBC, es el nivel mas bajo.

- **org.springframework.jdbc.object**

  Contiene clases que representan las consultas (queries) a los RDBMS, las actualizaciones (updates) y los procedimientos almacenados (stored procedures) en objetos reusables.

- **org.springframework.jdbc.support**

  Contiene la traducción de excepciones SQLException así como algunas clases de utilería.

4 CONFIGURACIÓN MAVEN

En este se utilizará Oracle XE, versión 11.2.0.2.0.

4.1 Definir el repositorio

```
<repositories>
  <repository>
    <id>codelds</id>
    <url>https://code.lds.org/nexus/content/groups/main-repo</url>
  </repository>
</repositories>
```

4.2 Configurar el driver

```
  <!-- Oracle JDBC -->
  <dependency>
      <groupId>com.oracle</groupId>
      <artifactId>ojdbc6</artifactId>
      <version>11.2.0.2.0</version>
  </dependency>
```

5 CONEXIÓN

5.1 Definición de una fuente de datos

En este caso se esta realizando con Oracle y el esquema eureka.

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:XE");
dataSource.setUsername("eureka");
dataSource.setPassword("admin");
```

5.2 Definición de un JdbcTemplate

Es necesario tener una fuente de datos.

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

6 CONSULTAS

6.1 Consultando un dato entero

```
String sql = "select count(*) from cuenta";
int rowCount = jdbcTemplate.queryForObject(sql,Integer.class);
System.out.println("Cantidad de cuentas: " + rowCount);
```

6.2 Consultando un dato double

```
String sql = "select sum(dec_cuensaldo) from cuenta";
Double saldo = jdbcTemplate.queryForObject(sql, Double.class);
System.out.println("Saldo total: " + saldo);
```

6.3 Consultando un String

```
String sql = "select vch_clienombre from cliente where chr_cliecodigo = ?";
Object[] parms = {"00001"};
String nombre = jdbcTemplate.queryForObject(sql, parms, String.class);
System.out.println("Nombre: " + nombre);
```

6.4 Consultando un registro de una tabla

Para este caso se utiliza la implementación de la interce `RowMapper` para pasar la fila actual de un `ResultSet` a un bean de tipo `Cliente`.

Primero se debe crear la clase entidad:

```
package pe.egcc.app.domain;

public class Cliente {

    private String codigo;
    private String paterno;
    private String materno;
    private String nombre;
    private String dni;
    private String ciudad;
    private String direccion;
    private String telefono;
    private String email;

    public Cliente() {
     }

    // Se debe crear los métodos getter y setter

}
```

Luego, se implementa la interface `RowMapper`:

```
package pe.egcc.app.mapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import pe.egcc.app.domain.Cliente;

public class ClienteMapper implements RowMapper<Cliente> {

    public Cliente mapRow(ResultSet rs, int i) throws SQLException {
        Cliente bean = new Cliente();
        bean.setCodigo(rs.getString("chr_cliecodigo"));
        bean.setPaterno(rs.getString("vch_cliepaterno"));
        bean.setMaterno(rs.getString("vch_cliematerno"));
        bean.setNombre(rs.getString("vch_clienombre"));
        bean.setDni(rs.getString("chr_cliedni"));
        bean.setCiudad(rs.getString("vch_clieciudad"));
        bean.setDireccion(rs.getString("vch_cliedireccion"));
        bean.setTelefono(rs.getString("vch_clietelefono"));
        bean.setEmail(rs.getString("vch_clieemail"));
        return bean;
  }

}
```

Finalmente, se ejecuta la consulta:

```
    String sql = "select chr_cliecodigo, vch_cliepaterno, vch_cliematerno, "
            + "vch_clienombre, chr_cliedni, vch_clieciudad, vch_cliedireccion, "
            + "vch_clietelefono, vch_clieemail "
            + "from cliente where chr_cliecodigo = ?";
    Object[] parms = {"00001"};
    Cliente bean = jdbcTemplate.queryForObject(sql, parms, new ClienteMapper() );
    System.out.println("Nombre: " + bean.getNombre());
    System.out.println("Paterno: " + bean.getPaterno());
    System.out.println("Materno: " + bean.getMaterno());
```

6.5 Consultando varios registros de una tabla utilizando RowMaper

Para este ejemplo se debe tener la clase de dominio `Empleado` y la clase `EmpleadoRowMapper`.

El resultado se obtiene en una lista de tipo: List<Empleado>.

```
String sql = "select chr_emplcodigo, vch_emplpaterno, vch_emplmaterno, "
        + "vch_emplnombre, vch_emplciudad, vch_empldireccion, vch_emplusuario "
        + "from empleado where vch_emplpaterno like ?";
Object[] parms = {"R%"};
List<Empleado> lista = jdbcTemplate.query(sql, parms, new EmpleadoMapper() );
for(Empleado bean: lista){
    System.out.println(bean.getCodigo() + " " + bean.getNombre() +
            " " + bean.getPaterno() + " " + bean.getMaterno());
}
```

6.6 Consultar todos los datos de una tabla utilizando Map

El resultado se obtiene en una lista del siguiente tipo: List<Map<String,Object>>.

```
String sql = "select int_movinumero nromov, dtt_movifecha fecha, "
        + "chr_tipocodigo tipo, dec_moviimporte importe "
        + "from movimiento where chr_cuencodigo = ?";
Object[] parms = {"00100001"};
List<Map<String,Object>> lista = jdbcTemplate.queryForList(sql, parms );
for(Map<String,Object> r: lista){
    System.out.println(r.get("nromov") + " " + r.get("fecha") +
        " " + r.get("tipo") + " " + r.get("importe"));
}
```

7 JDBCTEMPLATE Y ANOTACIONES

La clase JdbcTemplate es la pieza central del paquete de facilidades que ofrece Spring a la hora de trabajar con JDBC. Encapsula todos los detalles de conexión y gestión de recursos, permitiendo a los desarrolladores centrarse en la ejecución de sentencias.

7.1 Configuración

Para crear una instancia de una clase JdbcTemplate hace falta pasarle como parámetro en el constructor una referencia al DataSource. Así que partiendo de esta base, hay dos formas tradicionales de configurar una aplicación.

Lo primera es inyectar el DataSource y crear una instancia de JdbcTemplate:

```
private JdbcTemplate jdbcTemplate;

@Autowired
public void init(DataSource dataSource) {
    jdbcTemplate = new JdbcTemplate(dataSource);
}
```

La segunda es declarar un bean de tipo JdbcTemplate e inyectarlo:

```
<bean name="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg><ref bean="dataSource"/></constructor-arg>
</bean>


private JdbcTemplate jdbcTemplate;

@Autowired
public void init(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
}
```

JdbcTemplate es segura en entornos multihilos, y no debería representar ningún problema utilizar una única instancia.

Adicionalmente se puede heredar de la clase JdbcDaoSupport, que implementa directamente una propiedad de tipo JdbcTemplate, aunque requiere de igual forma que se le inyecte un DataSource.

7.2 Operaciones

Se muestra a continuación algunos ejemplos de la forma de trabajar con JdbcTemplate para realizar las operaciones más habituales contra base de datos.

Consulta sin parámetros que retorna un valor entero:

```
jdbcTemplate.queryForInt( "SELECT COUNT(1) FROM empresa" );
```

Consulta con parámetros que retorna un valor entero:

```
jdbcTemplate.queryForInt(
    "SELECT COUNT(1) FROM empleado WHERE id_empresa = ?",
    1L);
```

Consulta con parámetros de una cadena de texto:

```
jdbcTemplate.queryForObject(
    "SELECT nombre FROM empleado WHERE id_empleado = ?",
    new Object[]{1L},
    String.class)
```

Consulta con parámetros de una entidad:

```
Empresa empresa = jdbcTemplate.queryForObject(
    "SELECT id_empresa, nombre FROM empresa WHERE id_empresa = ?",
    new Object[]{1L},
    new RowMapper<Empresa>() {
        public Empresa mapRow(ResultSet result, int rowNum) throws SQLException {
            Empresa empresa = new Empresa();
            empresa.setIdEmpresa( result.getInt("id_empresa") );
            empresa.setNombre( result.getString("nombre") );
            return empresa;
        }
    }
);
```

Consulta sin parámetros de un listado de entidades:

```
List<Empresa> empresas = jdbcTemplate.query(
    "SELECT id_empresa, nombre FROM empresa",
    new RowMapper<Empresa>() {
        public Empresa mapRow(ResultSet result, int rowNum) throws SQLException {
            Empresa empresa = new Empresa();
            empresa.setIdEmpresa( result.getInt("id_empresa") );
            empresa.setNombre( result.getString("nombre") );
            return empresa;
        }
    }
);
```

Inserción de un registro:

```
jdbcTemplate.update(
    "INSERT INTO empresa(id_empresa, nombre) VALUES(?, ?)",
    1L, "UberShop");
```

Modificación de un registro:

```
jdbcTemplate.update(
    "UPDATE empresa SET nombre = ? WHERE id_empresa = ?",
    "Easy Dinner", 1L);
```

Borrado de un registro:

```
jdbcTemplate.update("DELETE FROM empresa WHERE id_empresa = ?", 1L);
```

Ejecución de una sentencia arbitraria:

```
jdbcTemplate.execute("CREATE TABLE nomina(id_nomina NUMBER(15) ... )");
```

Llamada a un procedimiento almacenado:

```
jdbcTemplate.update("CALL contrata_empleado(?)", 1L);
```

Como se observa, en ninguno de los casos hay que establecer conexiones ni gestionar recursos. No obstante, es importante tener en cuenta que aún pueden elevarse excepciones por sentencias mal escritas, parámetros incorrectos, o porque el objeto resultante de la ejecución no es el esperado según la nomenclatura del método.

7.2.1 NamedParameterJdbcTemplate

La clase NamedParameterJdbcTemplate es un wrapper sobre la clase JdbcTemplate que se instancia pasándole como argumento el DataSource en el constructor:

```
namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
```

Lo que pretende es resolver los problemas de tener que trabajar con caracteres marcadores de posición ("?") dentro de las sentencias SQL, permitiendo en su lugar utilizar nombres para los parámetros.

Los nombres de los parámetros se distinguen del resto de la sentencia antecediéndolos con un carácter de dos puntos (":"):

```
SqlParameterSource parameters = new MapSqlParameterSource("nombre", "Easy Dinner");
namedParameterJdbcTemplate.queryForInt(
    "SELECT id_empresa FROM empresa WHERE nombre = :nombre",
    parameters);
```

Una característica interesante es que se puede utilizar un objeto JavaBean con la clase BeanPropertySqlParameterSource, que construye automáticamente un Map con los nombres de las propiedades y sus correspondientes valores:

```
Empresa empresa = new Empresa();
empresa.setNombre("Easy Dinner");
SqlParameterSource parameters = new BeanPropertySqlParameterSource(empresa);
namedParameterJdbcTemplate.queryForInt(
    "SELECT id_empresa FROM empresa WHERE nombre = :nombre",
    parameters);
```

Otras clases similares ofrecidas por Spring permiten reducir la cantidad de código a escribir significativamente, como en las operaciones por batch para las actualizaciones masivas de registros por ejemplo:

```
SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch( empresas.toArray() );
namedParameterJdbcTemplate.batchUpdate(
    "INSERT INTO empresa(id_empresa, nombre) VALUES(:idEmpresa, :nombre)",
    batch);
```